Load-Balanced Pipeline Parallelism

Md Kamruzzaman Steven Swanson Dean M. Tullsen

Computer Science and Engineering
University of California, San Diego
{mkamruzz,swanson,tullsen}@cs.ucsd.edu

ABSTRACT

Accelerating a single thread in current parallel systems remains a challenging problem, because sequential threads do not naturally take advantage of the additional cores. Recent work shows that automatic extraction of pipeline parallelism is an effective way to speed up single thread execution. However, two problems remain challenging – load balancing and inter-thread communication. This work shows new mechanism to exploit pipeline parallelism that naturally solves the load balancing and communication problems. This compiler-based technique automatically extracts the pipeline stages and executes them in a data parallel fashion, using token-based chunked synchronization to handle sequential stages. This technique provides linear speedup for several applications, and outperforms prior techniques to exploit pipeline parallelism by as much as 50%.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors–Optimization

General Terms

Languages, Performance

Keywords

pipeline parallelism, load-balancing, chip multiprocessors, locality, compilers

1. INTRODUCTION

The number of cores per die on multicore processors increases with each processor generation. However, many applications fail to scale with the increased hardware parallelism. Several factors account for this, but this research is most concerned with key applications that are difficult to parallelize due to data dependences in the key loops, making the code highly sequential.

Previous work on *decoupled software pipelining* [18, 19, 21] addresses this problem and shows that fine-grained pipeline parallelism applied at the loop level can be very effective in speeding up some serial codes, including irregular codes like pointer chasing. In this case, the compiler automatically divides the loop into a set of pipeline stages (each stage can be sequential or parallel) and maps them to different cores to achieve parallel execution while still maintaining all the dependencies. However, several issues make the technique still challenging in practice. First, the cores often remain underutilized because of the imbalance in the pipeline, sacrificing performance and wasting energy. Second, the technique sacrifices the existing locality between stages, communicating data that was originally local across cores, again sacrificing both performance and power/energy. Finally, the technique typically works best with the number of threads at least equal to the number of stages, requiring the compiler to know a priori the number of cores available, and causing inefficient execution when the counts do not match.

This paper describes *load-balanced pipeline parallelism* (LBPP) which exploits the same pipeline parallelism as prior work, but assigns work to threads in a completely different manner, maintaining locality and naturally creating load balance. While prior pipeline parallelism approaches executes a different stage on each core, LBPP executes all the stages of a loop iteration on the same core, but achieves pipeline parallelism by distributing different iterations to the available cores and using token based synchronization to handle sequential stages. It groups together several iterations of a single stage, though, before it moves to the next stage.

LBPP is inherently load-balanced, because each thread does the same work (for different iterations). It maintains locality because same-iteration communication never crosses cores. The generated code is essentially the same no matter how many cores are targeted – thus the number of stages and the thread count are decoupled and the thread count can be determined at runtime and even change during runtime. Prior techniques must recompile to perform optimally with a different core count.

LBPP is a software only technique and runs on any cache coherent system. In this work, we present a compiler and runtime system that implements LBPP, and demonstrate how LBPP extracts maximum possible parallelism for any number of cores. We find that *chunking* is the key to making LBPP effective on the real machines. Chunking groups several iterations of a single stage together before the thread moves on to the next stage (executing the same chunk of iterations of that stage). Thus, chunking reduces the frequency of synchronization by clustering the loop iterations. Instead of synchronizing at each iteration, we only synchronize (and communicate) at chunk boundaries. Chunking appropriately allows us to maximize locality within a target cache.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC'13, November 17–21, 2013, Denver, Colorado, USA.
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503295.

In this work, we describe the implementation of LBPP under Linux and evaluate it on two multicore systems. We experiment with different microbenchmarks, focused on particular aspects of LBPP to characterize the technique better, and then apply it on a wide range of real applications (both regular and irregular). LBPP provides $1.7\times$ to $3.2\times$ speedup on AMD Phenom, and $1.6\times$ to $2.3\times$ on Intel Nehalem on average, depending on the core counts for individual loops. Considering application level speedup, some irregular applications see as large as $5.4\times$ speedup over the single thread execution. We also compare LBPP with decoupled software pipelining. LBPP outperforms decoupled software pipelining by 60% to 88% on the loop level for a particular core count (three cores) because of better load balancing and locality. The energy savings can be more than 50% for some applications.

The remainder of this paper is organized as follows: Section 2 discusses load-balanced pipeline parallelism in the context of prior work. Section 3 describes the technique in detail. Section 4 shows the compiler implementation. Sections 5 and 6 demonstrate our methodology and results, and Section 7 concludes.

2. RELATED WORK

There have been several works that seek to accelerate single thread execution on parallel hardware.

Decoupled software pipelining [18, 21] (DSWP) is a non-speculative technique that exploits pipeline parallelism on the loop level. Using dependence analysis, it automatically partitions an iteration of the loop into several stages and executes them in different processing units in a pipelined fashion to extract parallelism. Raman, et al. [19] propose parallel-stage DSWP that identifies some stages as parallel and executes them in a data parallel manner. LBPP adopts their concept of separating the loop iteration into a set of sequential stages or a mix of sequential and parallel stages, but the execution model is different. Unlike DSWP, which assumes hardware support for efficient synchronization and core to core communications, LBPP uses chunking to amortize synchronization overhead and naturally exploits locality to reduce inter-core communication.

DOACROSS [1, 6, 8] is another widely studied technique that distributes loop iterations like LBPP to extract parallelism, but it does not support arbitrary control flow inside the loop body. So, pointer chasing loops or loops with unknown bounds do not get advantages. In addition, DOACROSS does not distinguish between sequential and parallel stages, which is critical for load balancing and better utilization of processors. LBPP provides an efficient execution model (evaluated in the context of state of the art multicores) that ensures maximum possible parallelism for any number of cores even for irregular loops. Chen and Yew [4] focus on dependence graph partitioning to reduce synchronization overhead for DOACROSS loops. LBPP can use their algorithm as well to construct the pipeline. Loop distribution [12] splits a loop into several loops by isolating the parts that are parallel (have no cross iteration dependencies) from the parts that are not parallel and can leverage data parallelism.

Helper thread prefetching [11, 14, 15] is a special form of pipeline parallelism, because it pipelines memory accesses with the computation. In this case, prefetch threads can execute in either separate SMT contexts (targets the L1 cache) or in separate cores (targets shared cache) in parallel. Inter-core prefetching [11] adds thread migrations on top of helper thread prefetching to access the prefetched data locally. It uses chunking in a very similar manner to LBPP. LBPP can naturally treat prefetching as an extra stage in the pipeline.

Data Spreading [10] is another software technique that uses thread migrations to spread out the working set to leverage the aggregate cache space. LBPP, when executed with more than the expected number of cores, provides similar benefit.

Speculative multithreading [16, 23] ignores some of the dependence relationships to execute different parts of the serial execution in parallel and verify the correctness at runtime. Vachharajani, et al. [26] speculate on a few dependencies to extract more pipeline stages in DSWP. They use a special commit thread for the recovery process. Spice [20] uses value prediction of the loop live-ins to enable speculative threading. Huang, et al. [9] combine speculative parallelization with parallel stage DSWP. LBPP is non-speculative and orthogonal to the speculative techniques.

There have been several works on pipeline parallelism [2, 7, 13, 17]. Navarro, et al. [17] gives an analytical model for pipeline parallelism using queuing theory to estimate the performance. Bienia, et al. [2] use PCA analysis to characterize pipeline applications from data parallel applications. Communications between different pipeline stages is shown to be an important component. The cache-optimized lock-free queue [7] is a software construct that reduces communication overhead. Lee, et al. [13] propose hardware solutions to the queuing overhead. Chen, et al. [5] improves DSWP using a lock-free queue. LBPP does all its core to core communications through shared memory and does not require special queues.

Load balancing among threads has also received attention in the past. Work stealing [3] is a common scheduling technique where cores that are out of work steal threads from other cores. Feedback-driven threading [25] shows, analytically, how to dynamically control the number of threads to improve performance and power consumption. Suleman, et al. [24] propose a dynamic approach to load balancing for pipeline parallelism. Their technique tries to find the limiter stage at runtime and allocates more cores to it. They assume at least one core per stage. Sanchez, et al. [22] use task stealing with per-stage queues and a queue backpressure mechanism to enable dynamic load balancing for pipeline parallelism. LBPP is inherently load balanced.

3. LOAD-BALANCED PIPELINE PARALLELISM

Load-balanced pipeline parallelism exploits the pipeline parallelism available in the loops of applications and executes them in a data parallel fashion. It uses token-based synchronization to ensure correct execution of sequentially dependent code and reduces synchronization overhead using chunking. The iteration space is divided into a set of chunks and participating threads execute them in round-robin order. The technique can handle both regular and irregular codes.

Next, we give an overview of pipeline parallelism and then describe LBPP.

3.1 Pipeline parallelism

Pipeline parallelism is one of the three types of parallelism that we see in applications (vs. data parallelism and task parallelism). Pipeline parallelism works like an assembly line and exploits the producer-consumer relationship. There are several pipeline stages and each stage consumes data produced by previous stages. This maintains the dependences between different stages. The paral-

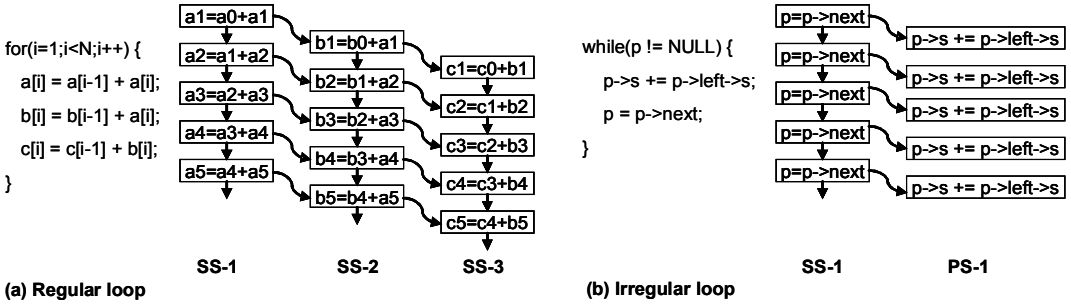

Figure 1: Examples of loop level pipeline parallelism for regular and irregular loops. The arcs show the dependency relationships.

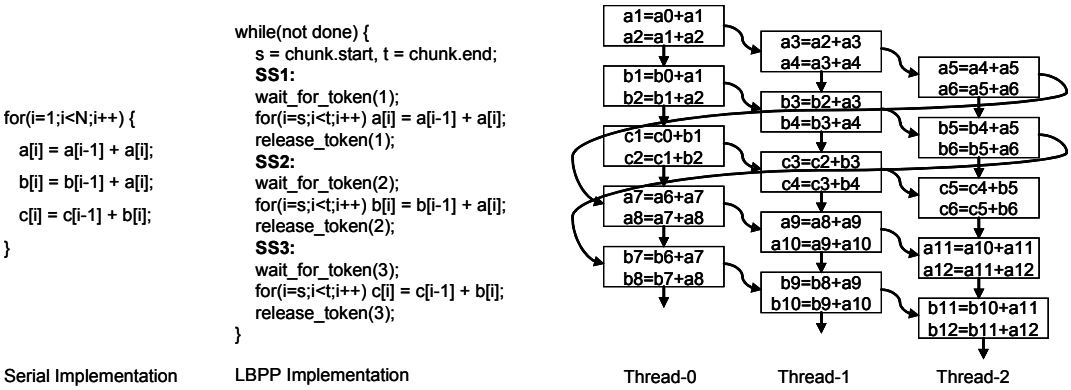

Figure 2: Implementation and execution of LBPP for a regular loop with a chunk size of 2 iterations that uses three threads.

lelism in this case is on the order of the number of stages. Pipeline parallelism can be coarse-grain or fine-grain.

In this work, we are interested in fine-grain pipeline parallelism extracted from loops in conventional code. Pipeline parallelism might exist in both regular and irregular loops. Figure 1 shows examples of pipeline parallelism in both types of loop. It also depicts the two different types of dependency relationships that exist between stages – *intra-iteration* dependency, and *cross-iteration* dependency. The former denotes the dependency between two stages in the same loop iteration whereas the latter represents a dependence to stages from previous iterations.

The regular loop in Figure 1(a) has three pipeline stages – SS1, SS2, and SS3. SS1 has no intra-iteration dependency (no incoming arcs from different stages), but has a cross-iteration dependency. SS2, and SS3 have both types of dependencies. All three stages in this case are *sequential* because of the cross-iteration dependency. We can map each stage to a separate core and execute the stages in parallel while still maintaining all the dependencies. Prior works describe this technique as decoupled software pipelining [21]. The maximum possible parallelism for this loop is $3\times$ assuming there are a sufficient number of iterations and all three stages are perfectly balanced.

Figure 1(b) shows the availability of pipeline parallelism in an irregular loop. In this case, SS1 is a sequential stage that does the pointer chasing. It has cross-iteration dependency but no intra-iteration dependency. Stage PS1 updates each node pointed to by *p* and depends only on SS1 computed in the same iteration. PS1 has no cross-iteration dependency and it is possible to compute different iterations concurrently for this stage, i.e. PS1 is a *parallel* stage. So, this irregular loop has a pipeline of one sequential and one parallel stage. We can use one thread for the sequential stage, and one or more for the parallel one to get a pipelined execution. Prior works address this form of execution as parallel-stage decoupled software pipelining (PS-DSWP) [19]. Assuming we have enough cores, the maximum possible parallelism for this loop is determined by the relative weight of the sequential stage because we must use one thread for the sequential stage, but can use as many as necessary to make the parallel stage non-critical.

3.2 Load-balanced pipeline parallelism

A pipelined loop (regular or irregular) has at least two stages including zero or more parallel stages. The traditional approach (DSWP or PS-DSWP) maps different stages to each core (or multiple cores, for parallel stages) and protects the intra-iteration dependency using synchronization. The cross-iteration dependency, on the other hand, is automatically satisfied by the program order. We use the term *traditional pipelining* (TPP for traditional pipeline parallelism) to refer to such execution.

Load-balanced pipeline parallelism executes all stages of a particular loop iteration in the same core, but distributes the iterations between cores in a manner more similar to conventional data parallel execution (which lacks the dependence edges). First, it splits the loop iterations into a set of *chunks*, and consecutive chunks of the same stage are executed on different cores. LBPP executes all the stages of a chunk sequentially in a single core. It will execute several iterations (a chunk) of stage 1, followed by the same iterations of stage 2, etc. Once all the stages are finished, the thread starts executing the next assigned chunk, starting again at the first stage. LBPP maps the chunks to threads in round robin order.

This correctly handles parallel stages, because they do not have

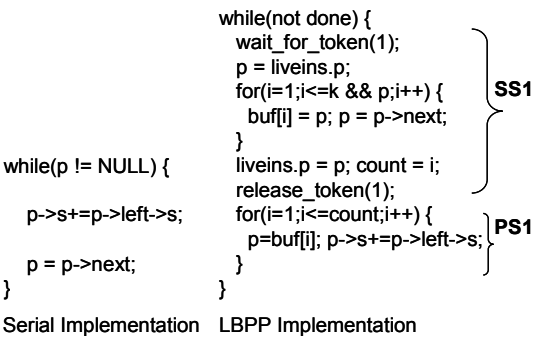

Figure 3: Implementation of LBPP for an irregular loop with a chunk size of k iterations.

any cross-iteration dependency, and the program order satisfies the intra-iteration dependency. However, sequential stages have cross-iteration dependencies. LBPP handles that with a simple token-based synchronization mechanism. There is a token for each sequential stage. Each thread waits for the corresponding token before executing a sequential stage. In that way, all of the iterations of a sequential stage execute in order – they execute in order on a single core within a chunk, and across chunks, the order is protected by token synchronization. The parallel stages do not require tokens and LBPP executes them immediately. Once a sequential stage finishes execution, the current thread hands over the token to the next thread. Thus, the tokens move between threads in round robin order and guarantee serial execution of a sequential stage.

LBPP obtains concurrency in two ways – (1) between stages, as threads are typically working on different stages at different times, and (2) within parallel stages, as any thread can enter the parallel stage without regard to whether other threads are in the same stage.

Figure 2 shows the LBPP implementation of the regular loop in Figure 1(a) using an artificially small chunk size of two iterations. Unlike traditional pipelining, all three threads in this case execute the same function. In TPP, we need to design separate functions for each pipeline stage, and then add synchronization.

LBPP manages irregular loops similarly. Figure 3 shows the LBPP implementation of the pointer chasing loop from Figure 1(b). The sequential stage, upon receiving the token, gets the start value of p from the previous thread. Then, it does the pointer chasing for k iterations and enqueues the pointers in a thread local buffer. After that, it forwards the current value of p to the next thread and releases the token. The parallel stage dequeues the sequence of pointers from the local buffer and does the update operation. Thus, the local buffer transfers data from the sequential stage to the parallel stage and serves the same purpose as software queues do in TPP. Note that queues/local buffers may not always be necessary. The regular loop mentioned above is an example of that.

The performance of LBPP depends heavily on the *chunk size*. We define chunk size as the approximate memory footprint of a chunk that includes both the memory accesses done by the original loop itself, and the additional memory accesses to the local buffer. Chunk size is highly correlated with the synchronization and data sharing overhead. Larger chunks better amortize the synchronization overhead (i.e., waiting for tokens) required for the sequential stages. The number of times that a thread needs to wait for the token is inversely proportional to the chunk size. Thus, when a sequential stage is critical (e.g., SS1 in Figure 1(b)), chunking minimizes the overhead of executing that stage in different cores.

Smaller chunks create more coherence traffic when there exist data sharing between consecutive iterations. Most cross-iteration communication will stay on a core – only when that communication crosses chunk boundaries will it move between cores. Therefore, larger chunks reduce communication. In Figure 2, for example, we see both true and false sharing. Both iteration 5 and 6 use $a5$, $b5$, and $c5$. On the other hand, there exists false sharing between iteration 4 and 6 when any of $a4, a6$ or $b4, b6$ or $c4, c6$ occupy the same cache line.

However, chunks cannot be arbitrarily large for two reasons. First, smaller chunks exploit locality better. Chunks that do not fit in the private caches will evict the shared data between two stages and will cause unnecessary long latency misses. Second, chunk size determines the amount of parallelism that can be extracted. In Figure 2, for three threads, chunks of 2 iterations gives parallelism of $2.25\times$ whereas 4 iterations per chunk gives parallelism of $1.8\times$ (there are 12 iterations in total). But, this gap diminishes when the loop iterates more. From the parallelism point of view, the size of the chunk relative to the number of loop iterations is important.

LBPP provides three key advantages over TPP – locality, thread number independence, and load balancing.

3.2.1 Locality

LBPP provides better locality compared to traditional pipelining. In TPP, data shared between stages within an iteration always crosses cores. Since pipelined loops always share data between stages, this is the dominant communication. In LBPP, this communication never crosses cores. TPP may avoid cross-core communication for loop-carried dependencies, but only if they go to the same stage. With LBPP, loop-carried dependencies only cross cores when they cross chunk boundaries, whether or not they are within a stage.

In Figure 1(a), the first two stages (SS1 and SS2) share a while the last two stages share b. Similarly, in Figure 1(b), the parallel stage PS1 uses p produced by SS1. Note that most of this sharing is read-write sharing, and all require expensive cache-cache transfers with TPP. The communication cost was also identified as a key bottleneck for traditional pipelining on real systems [19, 21].

In contrast, LBPP executes all stages of the same chunk in the same core. All the intra-iteration communication between stages happens within the same core. In Figure 3, PS1 will find p already available in the cache. The same thing happens for the regular loop in Figure 2. SS2 finds a while SS3 finds b in the private cache. LBPP can also exploit data sharing between distant stages, e.g. sharing between stage 1 and stage 4, etc.

For the example regular loop, assuming 3000 iterations, chunk size of 500, 8-byte values, and no false sharing, traditional pipelining would move 48,000 bytes for intra-iteration dependences while all loop carried communication is free. For LBPP, all intra-iteration communication is free, and we would move 144 bytes for loop-carried communication between chunks.

Chunking allows LBPP to target a certain level (either private or shared) of the cache hierarchy. By tuning the chunk size, LBPP can confine most memory operations to either the L1 cache or the L2 cache. In some sense, LBPP implements tiling between stages with the help of chunking.

3.2.2 Thread number independence

In Figure 2, there are three sequential stages. For TPP, the code created by the compiler would be very different depending on

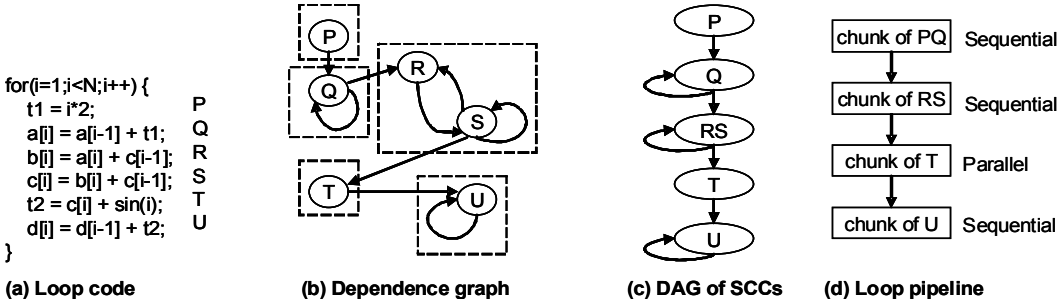

Figure 4: Different steps of constructing the pipeline for a loop.

whether it expected two cores or three cores to be available for execution. In LBPP, the code is the same in either case. Thus, LBPP decouples the code generation from the expected thread count, simplifying compilation and providing better adaptation to runtime conditions. We can use any number of threads irrespective of the number of pipeline stages for a loop, even including more threads than stages. For one thread, the loop executes the three stages sequentially. For two threads, LBPP achieves pipelining and extracts as much as $2\times$ parallelism when all three stages are of equal weight (TPP will suffer from imbalance). The parallelism improves to $3\times$ for three threads. With more threads, the parallelism does not improve, but we might get additional speedup because of data spreading [10]. Data spreading implies distributing the data of a sequential loop over several private caches, creating the effect of a single large, fast private cache. LBPP can emulate this mechanism just by growing the number of threads/cores until the working set fits in the caches, again without recompiling.

3.2.3 Load balancing

LBPP is inherently load balanced like the traditional loop-level data parallelism, because each thread does the same work, only on different iterations of the loop. Thus, performance tends to scale linearly until we reach the maximum performance. That is not typically the case with traditional pipelining, although it will approach the same peak performance eventually.

Assume, a pipeline has k sequential stages with execution times $S_1, S_2 \ldots S_k$ and SS_l is the largest stage with execution time, S_{mx}. We also assume that there are a large number of iterations and there is no synchronization overhead. Thus, if thread 1 releases a token at time t, thread 2 can grab that token and start executing at time t. Moreover, since there are enough iterations, we ignore the first and last few iterations where not all threads are active.

With these settings, there is an important observation. A thread will only wait for tokens when it needs to execute the bottleneck stage SS_l. That is because when it gets the token for SS_l, it would require the next token after S_{mx} time and by that time, the next required token will be available. So, the average execution time of an iteration will be $T + W$ where T is $\sum_{i=1}^{k} S_i$, and W is the waiting time for the token to execute SS_l.

We can compute W for n threads. A thread that just finishes SS_l will require the token to execute this stage again after $T - S_{mx}$ time. On the other hand, the token will be available after $(n-1) * S_{mx}$ time. So, W is $max(0, (n-1) * S_{mx} - (T - S_{mx}))$ or $max(0, n * S_{mx} - T)$. Thus, the average execution time of an iteration is $max(T/n, S_{mx})$ using n threads. The equation also holds when there are one or more parallel stages. In that case, T is $\sum_{i=1}^{k} S_i + \sum_{i=1}^{m} P_i$ where $P_1, P_2 \ldots P_m$ are the execution times for the parallel stages. The expected parallelism of such a pipeline using LBPP for n threads is given in the following equation:

$$LBPP_n = \frac{T}{max(T/n, S_{mx})} \tag{1}$$

Equation 1 shows that LBPP is load balanced and always achieves maximum possible parallelism for any thread count. The parallelism increases linearly until it reaches the theoretical max, i.e., T/S_{mx} (Amdahl's law). The minimum number of threads to gain the maximum parallelism is:

$$LBPP_{min} = ceil(T/S_{mx}) \tag{2}$$

Compared to that, TPP uses a single thread for each sequential stage, and one or more for each parallel stage. So, the minimum number of threads to achieve the maximum parallelism is:

$$TPP_{min} = k + \sum_{i=1}^{m} ceil(P_i/S_{mx}) \tag{3}$$

From Equation 2 and 3, $LBPP_{min}$ is always smaller than TPP_{min} unless all the stages have equal execution times. In that case, both values are the same. For example, assume a pipeline has 5 sequential stages with the execution times of 10, 15, 10, 20, and 5. TPP will require 5 threads to reach the maximum parallelism of $3\times$. However, LBPP will require only 3 threads to achieve that.

The load balancing becomes more evident in the presence of parallel stages. Assume a pipeline has 2 sequential and 2 parallel stages with the execution times of 10, 10, 40, and 40, respectively. With 4 cores, TPP extracts $100/40 = 2.5\times$ parallelism. However, LBPP gives $4\times$ parallelism and utilizes all threads 100%. LBPP provides more parallelism until we use 10 threads. In that case, both provide the maximum parallelism of $10\times$.

Looking closely, LBPP provides better parallelism because of better load balancing. In LBPP, the same parallel stage may execute across all cores. Similarly, several small sequential stages may execute on a single core. These things are not possible in TPP, because stages are tightly bound to threads.

Equation 1 also suggests that for LBPP, the parallelism is independent of the number of stages. This gives enormous flexibility while designing the pipeline. Load balancing allows us to add more pipeline stages or redesign the current stages and still ensures the maximum parallelism as long as T and S_{mx} do not change.

If we account for synchronization cost, there is one case where TPP may have an advantage. Once we have enough threads that a sequential stage becomes the bottleneck, TPP executes that stage

on a single core, while LBPP executes it across multiple cores. Both techniques will have some synchronization in the loop, but TPP may be able to avoid it when that stage only produces and no queuing is necessary (e.g., SS1 in our regular loop). However, for LBPP, it can be highly amortized with large chunks. In our experiments, this effect was, at worst, minimal, and at best, completely dominated by the locality advantages of LBPP.

4. LBPP IMPLEMENTATION

LBPP can be automatically implemented in the compiler and does not require any hardware support. Our implementation of LBPP follows very closely the implementation of traditional pipelining, at least in the initial steps. In fact, any traditionally pipelined loop can be automatically converted to take advantage of LBPP.

There are four steps to apply LBPP – DAG construction, pipeline design, adding synchronization, and chunking.

4.1 DAG construction

In this step, we construct the dependence graph for a loop using the same methodology described in previous work [18, 19]. The dependence graph includes both data and control dependence. Figure 4(b) shows the dependence graph for the loop on the left side. An arc from node x to node y denotes that node y must execute after the execution of x.

Next, we identify the strongly connected components (SCC) in the dependence graph. Nodes in an SCC have cyclic dependencies, and cannot be pipelined. Executing all the nodes of an SCC in the same thread maintains the chain of dependency. The rectangular boxes in Figure 4(b) represent the SCCs. If we consider each SCC as a single node, we get a directed acyclic graph (DAG). This is called the *DAG of SCCs*. We can do a topological ordering for the DAG so that all edges are either self edges or go forward to subsequent nodes, i.e., there will be no backward edges. Figure 4(c) shows the DAG for this example loop.

In a DAG, the self arc represents cross-iteration dependency whereas forward arcs represent intra-iteration dependency. We can pipeline all the nodes of a DAG by making each node a different pipeline stage. This will satisfy all the dependencies. The nodes that do not have self arcs (P, T in Figure 4(c)) can be executed in parallel as well. These become the parallel stages in a pipeline. The others (Q, RS, U) become sequential stages.

4.2 Pipeline design

LBPP is load balanced and independent of how many threads will be used at runtime. This makes pipeline design easy and flexible.

From Section 3.2.3, we need to minimize both T (the sum of all stage execution times) and S_{mx} (the maximum of all stage execution times) for maximum parallelism. The DAG construction automatically minimizes S_{mx}. However, using all DAG nodes as separate stages may not minimize T. There are overheads for each additional stage. These include the synchronization overhead for sequential stages, decoupling overhead (loading and storing data from local buffers), reduced instruction level parallelism, and other software overheads. Chunking automatically amortizes most of these overheads. To reduce them further, we can cluster stages as long as S_{mx} does not increase.

For clustering, we first collapse simple producer stages with corresponding consumer stages. In Figure 4(c), stage P computes $i*2$ used by stage Q. Keeping P as a separate stage will cause a store operation (enqueue $i*2$) in P, and a load operation (deque $i*2$) in Q. Collapsing P with Q will remove these memory operations and the overheads for using one extra stage. This also compacts Q because the load operation is more expensive than computing $i*2$ in most systems. As a rule of thumb, we collapse a stage unless it does enough computation that it is equivalent to doing at least three memory operations.

```
while( node != root ) {
  while( node ) {
    if( node->orientation == UP )
      node->potential=node->basic_arc->cost+node->pred->potential;
    else {
      node->potential=node->pred->potential–node->basic_arc->cost;
      checksum++;  }
    tmp = node;  node = node->child; }
  ..............
}
```

Figure 5: Nested loop of *refresh_potential* function of *mcf*.

Next, we try to coalesce parallel stages similar to that described in the earlier work [19]. We combine two parallel stages PS1 and PS2 when there is no intervening sequential stage, SS such that there is an arc from PS1 to SS and SS to PS2 in the DAG of SCCs. We continue this process iteratively unless no more coalescing is possible. This process of coalescing does not change S_{mx}. So, from Equation 1, there is no negative impact on the level of parallelism.

LBPP works well if we design the pipeline using above steps. We can further optimize the pipeline by collapsing the stages whose combined execution time does not exceed S_{mx}. This requires to estimate the stage execution times, which can be done by using a cost model, or by profiling. In this work, we use a simple cost model that counts different operations and takes an weighted sum for the estimation. The model assumes a large weight for function calls or when the stage itself is a loop. Note that for LBPP, we do not need a perfect partitioning of stages, but we should remove simple stages (easy to identify) that benefit less than the overheads induced. So, a precise execution time for complex stages (having function calls or loops) is not necessary. Next, we use a simple greedy algorithm for the merging of stages, because LBPP does not require the optimal solution like TPP. For example, if the execution times are 60, 40, 20, 30, 20, and 10 – both 60, 60, 60 and 60, 40, 50, 30 are equally good solutions.

The output of the pipeline design step is a sequence of sequential and parallel stages. Figure 4(d) shows the pipeline for our example loop. We add necessary memory operations to transfer data from one stage to another stage. In our example, we buffer $t2$ in stage T and use that in stage U.

4.3 Chunking

Our compilation framework adds necessary code for chunking the pipeline stages. For a particular chunk size, we compute the number of iterations per chunk (say k) using the memory footprint of a loop iteration. LBPP executes k iterations of a stage before going to the next stage.

Figures 2 and 3 show the chunking of both regular and irregular loops. For regular loops (loop count is known), each thread independently identifies the chunks for themselves and execute those chunks. The threads exit when there are no more chunks to execute. For irregular loops, LBPP uses a sequential stage (SS1 in Figure 2) that catches the loop termination in one thread and for-

CPU components	Intel Nehalem, AMD Phenom
CPU Model	Core i7 920, Phenom II X6 1035T
Number of cores	4, 6
L1 cache	32-Kbyte 4 cycles, 64-Kbyte 3 cycles
L2 cache	256-Kbyte 10 cycles, 512-Kbyte 15 cycles
L3 cache	8-Mbyte shared inclusive 46 cycles, 6-Mbyte shared exclusive 45 cycles
DRAM hit latency	190-200 cycles, 190-200 cycles
Cache to cache transfer latency	42 cycles, 240 cycles

Table 1: Microarchitectural details of the Intel and AMD systems.

wards that information to other threads. LBPP automatically packs and unpacks necessary liveins for all the coordinations.

In this work, we statically compute the approximate memory footprint of a loop iteration. This is easy for loops that do not have inner loops. For nested loops, the iteration memory footprint of the outer loop depends on the number of iterations of the inner loop. Figure 5 shows an example of that. In this case, the number of iterations of the inner loop is not fixed, so the memory footprint of the outer loop iteration changes dynamically. If we apply chunking to the outer loop and use a fixed value of k, chunks will not be of even size and load distribution between threads will be imbalanced. We solve this problem by adding a lightweight sequential stage that counts the number of inner loop iterations. Thus, we can compute a more accurate value of k and do better load distribution – we terminate a chunk based on the inner loop count rather than the outer loop count. Note that in LBPP, adding a lightweight stage is inexpensive since it does not require a new thread. The technique also works for multiple inner loops. In fact, it can also be used for load balancing of pure data parallel loops.

In Section 6, we will examine other metrics besides memory footprint for determining chunk size.

4.4 Adding synchronization

In the final step, we add the synchronization operations (waiting for tokens and releasing tokens) to the sequential stages. The compilation framework creates the necessary tokens at the beginning of the program. At the start of a pipelined loop execution, the framework assigns all the corresponding tokens to the thread that executes the first chunk. When a thread completes a sequential stage, it passes the token to the next thread. There is also a barrier at the end of each pipelined loop to make all changes visible before proceeding to the computation following the loop.

5. METHODOLOGY

We choose two state of the art systems from different vendors to evaluate LBPP. Table 1 summarizes the important microarchitectural information of both systems. The systems run Linux 2.6. We compile all our codes using GCC version 4.5.2 with "-O3" optimization level. We keep hardware prefetching enabled for all of the experiments. For power measurements, we use a "Watts up? .Net" power meter that measures the power at the wall and can report power consumption in 1-second interval with $\pm 1.5\%$ accuracy.

We apply LBPP on loops from a diverse set of applications chosen from different benchmark suites – Spec2000, Spec2006, Olden,

```
for(i=1;i<N;i++) {
  SS1:
  a[i] = sin(a[i-1]+a[i]+1);
  SS2:
  b[i] = sin(b[i-1]+a[i]+1);
  SS3:
  c[i] = sin(c[i-1]+b[i]+1);
  SS4:
  d[i] = sin(d[i-1]+c[i]+1);
  SS5:
  e[i] = sin(e[i-1]+d[i]+1);
}
(a) mb_load
```

```
for(i=2;i<N;i++) {
  SS1:
  a[i] = (a[i-2] + a[i-1] +
          a[i]) / 3.0;
  PS1:
  b[i] = sin(a[i]) * cos(i);
  SS2:
  c[i] = (c[i-1] + a[i]   +
          b[i]) / 3.0;
  PS2:
  d[i] = sin(c[i]) + M_PI;
}
(b) mb_ubal
```

```
t = start_node;
while(t != stop_node)
{
    SS1:
    tmp = t;
    t = t->next;
    SS2:
    s += tmp->left->a;
}
(c) mb_local
```

Figure 6: Source code of the microbenchmarks that explore different aspects of LBPP.

SciMark2. We also pick two important irregular applications (also used in previous work) – *ks* (Kernighan-Lin graph partitioning algorithm), and *otter* (an automated theorem prover). We exclude some benchmarks whose key loops have either one parallel or sequential stage (no pipelining possible). In addition, our compilation framework currently only handles C code, further limiting candidate benchmarks.

Table 2 shows the loops of interest, corresponding function names, and the description of the benchmark. We select a loop if it is not parallel and contributes at least 10% to the total execution time, when executed serially with the reference input. Table 2 also shows each loop's contribution to the total execution time in the AMD system, and the pipeline structure identified by the compiler. There are four pointer chasing irregular loops – *refresh_potential*, *FindMaxGpAndSwap*, *BlueRule*, and *find_lightest_geo_child*. Most of these loops have inner loops.

6. RESULTS

We evaluate LBPP in several steps. First, we use a set of simple microbenchmarks to explore the potential and different characteristics of LBPP. Then, we evaluate its impact on the real applications described in Section 5. Finally, we study LBPP's impact on the power and energy consumption.

For all of our experiments, we also implement traditional pipelining (TPP) as efficiently as possible. This is straightforward because the compiler-extracted pipeline structure and even the optimization of the stages (coalescing, etc.) are the same for both techniques. In fact, both LBPP and traditional pipelining use similar synchronization constructs.

6.1 Microbenchmarks

We design several simple microbenchmarks whose pipeline structures are easy to recognize. They capture a variety of characteristics including the amount of data transfer between stages, the organization of sequential and parallel stages, and the relative weight of stages. Figure 6 shows the source code and pipeline structures of the three microbenchmarks. All of them run many iterations and have large working sets unless specified otherwise. The speedup numbers given in this section and afterwards are all normalized to the single thread execution with no pipelining.

6.1.1 Load balancing and number of stages

Microbenchmark *mb_load* has five sequential stages and the pipeline is mostly balanced. Adjacent stages share data, but the amount of data sharing compared to the computation per stage is small, because the *sin* function is expensive.

Function name	Benchmark name	Suite	Contribution	Pipeline structure
match, train_match	art	Spec2000	77%	ss1 ps1
f_nonbon	ammp	Spec2000	12%	ss1 ps1 ss2
smvp	equake	Spec2000	68%	ss1 ps1 ss2
SetupFastFullPelSearch	h264ref	Spec2006	30%	ss1 ss2 ps1
P7Viterbi	hmmer	Spec2006	99%	ps1 ss1
LBM_performStreamCollide	lbm	Spec2006	99%	ps1 ss1
refresh_potential	mcf	Spec2006	19%	ss1 ps1 ss2
primal_bea_mpp	mcf	Spec2006	61%	ps1 ss1
FindMaxGpAndSwap	ks	Graph partitioning	100%	ss1 ps1 ss2
BlueRule	mst	Olden	77%	ss1 ps1 ss2
find_lightest_geo_child	otter	Theorem proving	10%	ss1 ps1 ss2
SOR_execute	ssor	SciMark2	99%	ps1 ss1

Table 2: List of benchmarks explored in our experiments. Here, *ss* represents sequential stage where *ps* stands for parallel stage.

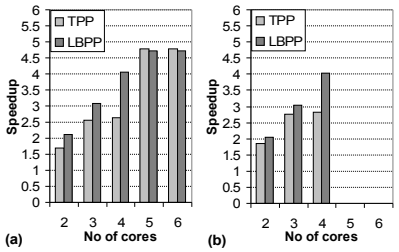

Figure 7: Scalability of *mb_load* – (a) Phenom (b) Nehalem

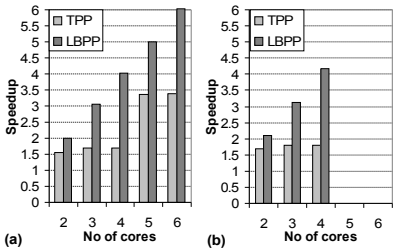

Figure 8: Scalability of *mb_ubal* – (a) Phenom (b) Nehalem

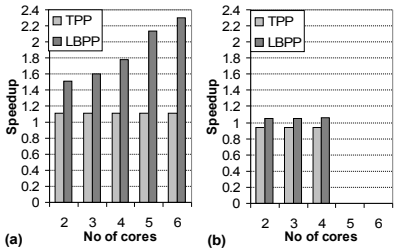

Figure 9: Scalability of *mb_local* – (a) Phenom (b) Nehalem

We apply LBPP and TPP on *mb_load* for different number of threads. For LBPP, we use the same pipeline structure for all thread combinations. However, the 5-stage pipeline given above is only applicable for five threads when we use TPP. So, we make the best possible partition of the stages into 2, 3, and 4 groups to run with 2, 3, and 4 threads, respectively, but a balanced partitioning is not possible. Figure 7 shows the speedup using both techniques on the two experimental systems. We use several chunk sizes for both LBPP and TPP and show the data for the best chunk size. Chunking lets TPP use synchronization and enqueuing once for a group of iterations, like LBPP. Section 6.2 gives more information about choosing the chunk size.

LBPP improves performance linearly in both systems. The maximum possible parallelism for this loop is $5\times$. LBPP provides $4.7\times$ speedup for five cores in the AMD system and $4\times$ for four cores in the Intel system. From Equation 1, the parallelism will not increase once it gets limited by the largest sequential stage. So, in the AMD machine, the speedup stays the same beyond five cores.

Figure 7 also demonstrates the importance of decoupling the pipeline design and the number of threads to be used. TPP gives the same speedup as LBPP for five cores. However, it loses as much as 35% ($4.1\times$ vs. $2.7\times$) and 30% ($4\times$ vs. $2.8\times$) for four threads in the AMD and Intel system, respectively, despite using the optimal partitioning of the stages. Lack of load balancing accounts for this. With four partitions of the five similar stages, the largest partition is twice as big as the smallest one, and makes one core overloaded. LBPP also outperforms traditional pipelining while using 2 or 3 cores.

6.1.2 *Unbalanced stages*

LBPP effectively handles the pipeline where there are multiple parallel stages and the stages are not balanced. The microbenchmark *mb_ubal* has two parallel stages and two sequential stages of relative weight of 60, 30, 5, and 5, respectively. For TPP, we use the best possible partitioning when there are less than four threads, and assign the extra threads to the parallel stages when there are more.

Figure 8 shows the performance for this pipelined loop. LBPP provides linear performance improvements for all thread combinations in both machines. The sequential stages here are much shorter than the parallel stages. LBPP does not bind stages to threads and dynamically assigns stages to threads. With LBPP, it is possible that all threads are executing the largest parallel stage PS1 at a time, where TPP only gets stage parallelism when it can explicitly assign multiple threads to a stage. LBPP provides $6\times$ speedup for six cores in AMD Phenom and $4.2\times$ for four cores in Intel Nehalem.

In this case, TPP suffers from load imbalance, because the smaller stages occupy cores and sit idle waiting for the data from the largest stage, PS1. This results in performance loss of 55% in Nehalem, and 58% in Phenom for four cores (each stage gets its own core). The situation does not improve much even when we use extra threads for PS1 and PS2. The performance loss is 44% for six cores in the AMD machine. In fact, from Equation 3, for any thread count of less than 20, LBPP will outperform TPP.

6.1.3 *Locality*

Microbenchmark *mb_local* explores the locality issue in pipelining. This irregular loop uses a 1 MB working set and executes mul-

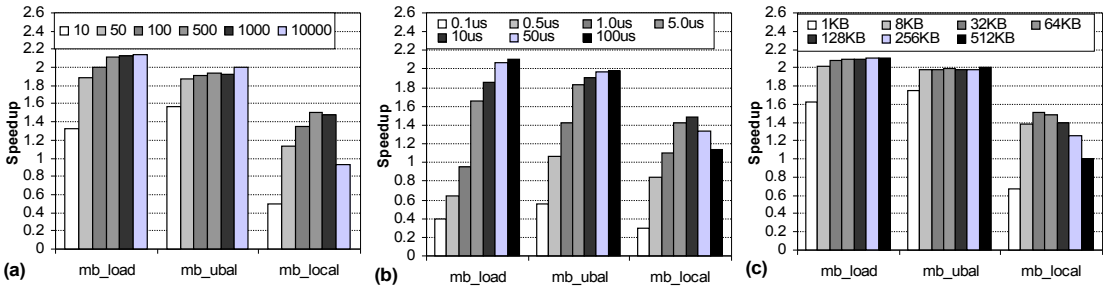

Figure 10: Performance impact in LBPP of different chunk sizes in the AMD Phenom system – (a) Iteration based, (b) Execution time based, (c) Memory footprint based.

tiple times. Thus, the same data is reused again and again. Stage SS1 does the pointer chasing and supplies pointers to SS2 to do the accumulation. So, there exists significant data sharing between the two sequential stages.

Figure 9 compares the performance between LBPP and TPP. Here, two cores are enough to exploit the available parallelism for both techniques. Yet LBPP outperforms TPP by 36% and 11% for Phenom and Nehalem, respectively, because it exploits locality. The pointers traversed by SS1 stay in the local cache for LBPP. However, for TPP, it requires expensive cache to cache transfers (Table 1) to send the pointers from SS1 to SS2. In Nehalem, this causes 16% more references to the last level cache. The loss is more prominent in Phenom than it is in Nehalem. This directly correlates with the latency of cache to cache transfers, which is much higher in Phenom (Table 1), making locality more critical.

Figure 9 also shows LBPP's ability to exploit data spreading [10]. Nehalem does not experience this, because the working set does not fit in the combined L2 cache space. For Phenom (with more cores and larger L2 caches), the working set gets distributed and fits in the aggregate L2 cache space as we add more cores. This does not increase parallelism, but reduces average memory access time. The effect improves LBPP's performance up to $2.3\times$ while using six cores, even though there are only two sequential stages.

6.2 Impact of chunk size

Chunking is a critical component for LBPP, because it amortizes the overhead due to data sharing and synchronization. So far, we have used chunking based on the approximate memory footprint of an iteration. We can also do chunking using the number of iterations directly, or using the execution time of an iteration (using expected iteration latency to compute a chunk iteration count). Thus, chunk size can be a simple iteration count or execution time (in μs) other than the memory footprint.

Figure 10 shows the performance for our microbenchmarks using the three chunking techniques. We give the results for Phenom and use two cores. We vary the chunk size from 1 to 10000 iterations, $0.1\mu s$ to $100\mu s$, and 1KB to 512KB for the iteration based, execution based, and footprint based chunking, respectively.

Iteration based chunking is the simplest. Figure 10(a) explains the need for chunking. Without chunking (chunk size of one iteration in the graph), the performance seriously suffers. The loss can be as high as 93% compared to the single threaded execution that uses no pipelining. Using 10 iterations per chunk is also not sufficient. The tightest loop, *mb_local* cannot amortize the synchronization overhead and shows negative performance. We see significant performance improvements for 50 iterations and it improves further for 500 iterations across all microbenchmarks. Thus, LBPP requires significant computation per chunk to keep the synchronization overhead under control.

Figure 10(b) explains the correlation between the chunk execution time and the synchronization overhead. LBPP cannot amortize the overhead when the chunks execute for $0.5\mu s$ or less. LBPP amortizes the overhead well and reaches close to $2\times$ speedup when the chunk size is at least $10\mu s$. This gives an effective bound for choosing the chunk size. Out of three, *mb_load* shows more sensitivity to chunk size, because it has five sequential stages and it does more synchronization per iteration than the others do.

Memory footprint based chunking handles the locality issue. We can tune the chunk size to target a particular level of the cache hierarchy. Memory footprint also strongly correlates with the execution time. Figure 10(c) shows that chunk sizes starting from 8KB work well across our microbenchmarks. It also shows that by keeping the chunk size limited to 32KB or 64KB, we can confine the data transfers from SS1 to SS2 for *mb_local* in the L1 cache and get maximum benefit.

Chunking also helps traditional pipelining. In this case, the primary advantage is to reduce the instances of synchronization. So, we implement chunking for TPP as well for fair comparison.

Chunk size selection.

The above experiments show the flexibility of using chunk size for LBPP. Memory footprint based chunking is particularly effective for compilers, because it is possible to approximate the iteration footprint statically in most cases, and a wide range of chunk sizes between 8KB and 256KB work well. Compilers can use any default chunk size (e.g., 32KB, or the size of the L1 cache). Chunk size is also a runtime parameter, and can be tuned further for a particular application by profiling.

6.3 Application Performance

This section describes the performance of LBPP on real applications. Table 2 gives detailed information about the loops and their corresponding pipeline structure. All the loops have at least one parallel stage and one sequential stage. Thus, the mix of parallel and sequential stages is very common.

Figure 11 shows the loop level speedup of LBPP for different number of cores in the AMD Phenom machine. We normalize the performance using both single thread execution (left graph), and using TPP with the same number of cores (right graph). For TPP, when there are more cores than the number of stages, we apply the

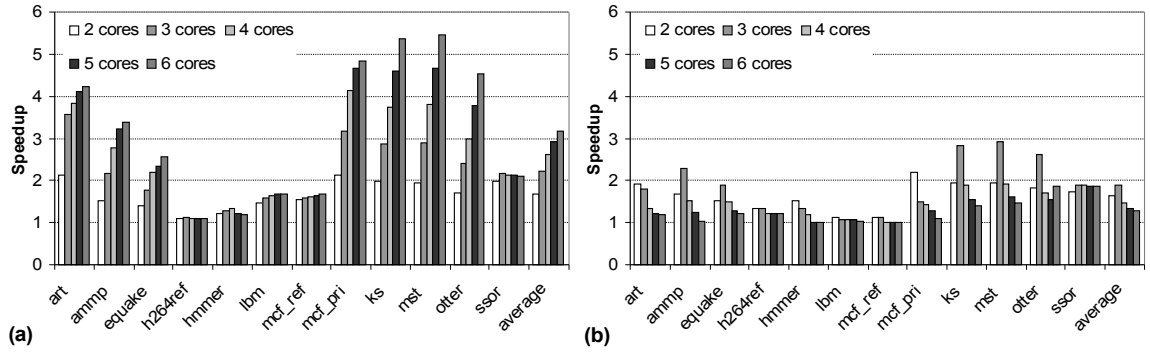

Figure 11: Performance of LBPP on sequential loops from real applications using different core counts in AMD Phenom – (a) normalized to single thread, (b) normalized to TPP.

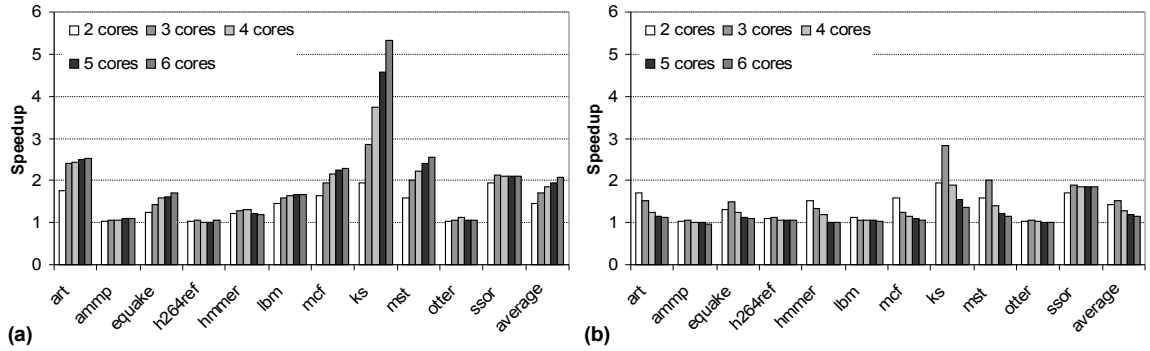

Figure 13: Application level speedup in AMD Phenom – (a) normalized to single thread, (b) normalized to TPP.

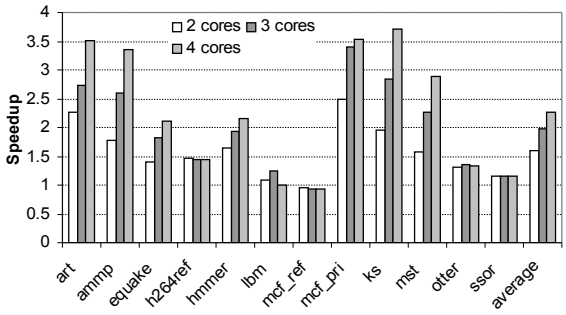

Figure 12: LBPP performance on the Intel Nehalem for real application loops – normalized to single thread.

extra cores to the parallel stage. We try with the seven footprint based chunk sizes given in Figure 10(c) and show the result for the best chunk size for both LBPP and TPP.

LBPP provides significant performance improvements across all loops, an average speedup of $1.67\times$ for just two cores. The pointer chasing loop in the complex integer benchmark, *mcf* shows $1.57\times$ speedup justifying the importance of loop level pipelining. Some of the loops (e.g., *ks*, *mst*, etc.) show linear scalability. The speedup can be as high as $5.5\times$ in some cases. On average, the performance improves from $1.67\times$ to $3.2\times$ when we apply six cores instead of two cores. Some of the loops do not scale well, because the parallel stage is not large enough and the sequential stages start to dominate according to Equation 1. The loops from *equake* and *lbm* have larger parallel stages, but lose some scalability due to the data sharing and off chip bandwidth limitation, respectively. The *equake* loop has significant write sharing across different iterations, enough that the communication across chunk boundaries makes an impact. The *lbm* loop is bandwidth limited and cannot exploit additional computing resources.

LBPP outperforms traditional pipelining in almost all cases. The difference is much bigger for smaller core counts, because of the load balancing and locality issues. LBPP outperforms TPP by 65% for two cores, and by 88% for three cores on average. Traditional pipelining closes the gap for higher core counts (5 to 6 cores). For six cores, LBPP wins by 27%. In that case, there are enough cores for all stages and load balancing does not remain a critical issue. This reduces the locality issue to some extent, especially when the parallel stage gets data from another stage. TPP still moves large amounts of data, but with more cores, it is better able to hide the communication. We see this effect in the *refresh_potential* loop of *mcf*, for example.

As a further exploration of scalability, we pick three loops (*ks*, *mst*, and *ssor*) where LBPP outperforms TPP by big margin and then compare the performance in a dual socket 12-core system to check whether TPP can match LBPP for even larger core counts. Using 12 cores, LBPP outperforms TPP by 5%, 15%, and 7% for *ks*, *mst*, and *ssor*, respectively. So, LBPP always remains a better choice even if we add lot more cores to solve the load balancing issue.

We also evaluate LBPP and TPP in our Nehalem system (Figure 12). LBPP provides $1.6\times$ and $2.3\times$ speedup on average us-

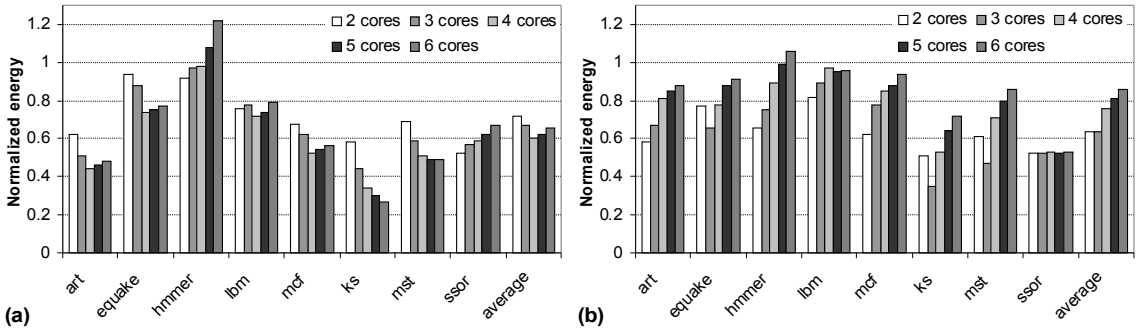

Figure 14: Energy consumption in Phenom – (a) normalized to single thread, (b) normalized to TPP. Here, lower is better.

ing two and four cores, respectively. This is 50% and 26% higher than what traditional pipelining offers. Thus, LBPP performs well across different architectures.

The results so far show the performance of the individual loops. Figure 13 describes the impact on the application level for the AMD machine, since those loops only represent a fraction of total execution time. The application performance varies depending on the loop's contribution to the total execution time, but also on how the loop interacts with the rest of the program. For example, the serial part might suffer more coherence activity because of the data spreading by the pipelined loop. LBPP provides $5.3\times$ and $2.6\times$ speedup for two irregular applications *ks* and *mst*, respectively. For *mcf*, it is $2.3\times$. However, otter sees only 7% improvements even though the loop level speedup is $4.5\times$. Overall, even at the application level LBPP speedups are high, including outperforming TPP by 42% and 51% for two and three cores, respectively.

For some of these applications like *ammp*, most of the part is parallel. We do not apply LBPP (or TPP) on parallel loops, and execute them sequentially. So, the loop level performance improvement by LBPP does not quite reflect in the application level speedup (9% speedup for six cores). Assuming, linear scalabilty of the parallel part of *ammp*, LBPP provides $5.5\times$ speedup for six cores.

6.4 Energy Considerations

We measure the total system power using a power meter to understand the power and energy tradeoff for LBPP. The meter reports the power consumption in 1-second intervals, prohibiting power measurement at the loop level, because most of the loops execute for much shorter than 1 second at a time. Thus, we pick the benchmarks where the selected loops contribute at least 60% to the total run time and dominate the power consumption.

Figure 14 shows the normalized energy consumption in the AMD systems for different numbers of cores. LBPP provides significant energy savings over single thread execution across all core counts, due to decreasing execution times. LBPP beats TPP for all core counts because of better load balancing and faster execution. The energy savings is 36% on average for two and three cores. The improvement in locality makes a significant difference. This can be seen at large core counts where the difference in energy is far higher than the difference in performance. In some cases, we even observe LBPP consuming less *power* than that of TPP even though it executes faster. For *mst* and *equake*, the power savings is around 8% and 7%, respectively, while using three cores.

7. CONCLUSION

This paper describes Load-Balanced Pipeline Parallelism. LBPP is a compiler technique that takes advantage of the pipelined nature of sequential computation, allowing the computation to proceed in parallel. Unlike prior techniques, LBPP preserves locality, is naturally load-balanced, and allows compilation without a priori knowledge of the number of threads. LBPP provides linear speedup on a number of important loops when prior techniques fail to do so.

LBPP works by chunking, or executing a number of iterations of a single stage, before moving on to the next stage. For a sequential stage, a synchronization token is sent to another thread to continue with the next chunk. In this way, intra-iteration communication is always local, and even cross-iteration communication is minimized. Also, because all threads execute all stages, it is naturally load-balanced.

LBPP outperforms prior pipeline parallel solutions by up to 50% or more on full applications, especially for lower thread counts. It provides even more striking energy gains, by reducing both runtimes and decreasing expensive cache-to-cache transfers.

Acknowledgments

The authors would like to thank the anonymous reviewers for many useful suggestions. This work was supported by NSF grants CCF-1018356 and CCF-0643880 and by SRC Grant 2086.001.

8. REFERENCES

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. wen Tseng. An overview of the suif compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, 1993.

[2] C. Bienia and K. Li. Characteristics of workloads using the pipeline programming model. In *Proceedings of the ISCA*, 2012.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5), Sept. 1999.

[4] D.-K. Chen and P.-C. Yew. Statement re-ordering for doacross loops. In *Proceedings of the ICPP - Volume 02*, 1994.

[5] W. R. Chen, W. Yang, and W. C. Hsu. A lock-free cache-friendly software queue buffer for decoupled software pipelining. In *ICS*, 2010.

[6] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *ICPP'86*, 1986.

[7] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the PPoPP*, 2008.

[8] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An overview of the fortran d programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, 1991.

[9] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the CGO*, 2010.

[10] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Software data spreading: leveraging distributed caches to improve single thread performance. In *Proceedings of the PLDI*, 2010.

[11] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the ASPLOS*, 2011.

[12] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of the Supercomputing*, 1990.

[13] S. Lee, D. Tiwari, Y. Solihin, and J. Tuck. Haqu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *Proceedings of the HPCA*, 2011.

[14] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proceedings of the PLDI*, 2002.

[15] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the ISCA*, 2001.

[16] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. In *International Conference on Supercomputing*, 1998.

[17] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical modeling of pipeline parallelism. In *Proceedings of the PACT*, 2009.

[18] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the International Symposium on Microarchitecture*, 2005.

[19] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO*, 2008.

[20] E. Raman, N. Va hharajani, R. Rangan, and D. I. August. Spice: speculative parallel iteration chunk execution. In *CGO*, 2008.

[21] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the PACT*, 2004.

[22] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *PACT*, 2011.

[23] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the ISCA*, 1995.

[24] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. In *Proceedings of the PACT*, 2010.

[25] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. In *Proceedings of the ASPLOS*, 2008.

[26] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the PACT*, 2007.